Generics

CS1812/13: Object Oriented Programming II
Dr Reuben Rowe and Dr Matteo Sammartino
(based on slides by Dr Johannes Kinder)

Polymorphism

- Operations that apply to values of multiple types are *polymorphic*
- Example: in most languages, "+" accepts values of type integer or float

```
3 + 4
```

```
3.0 + 4.0
```

- Two kinds of Polymorphism in Java:
  - Ad-hoc Polymorphism
  - Parametric Polymorphism

Ad-hoc Polymorphism

In ad hoc polymorphism, a function is made to accept multiple parameter types by defining a customised version for each type

- Java supports ad-hoc polymorphism by **overloading**

```
public static void printQuoted(String s) {
    System.out.println("\"" + s + "\"");
}

public static void printQuoted(int i) {
    System.out.println("\"" + i + "\"");
}

public static void main(String[] args) {
    printQuoted("Hello"); // prints "Hello"
    printQuoted(5);       // prints "5"
}
```

Parametric Polymorphism

In parametric polymorphism, a data structure or function is defined for multiple types by making the value type an abstract parameter

- With parametric polymorphism, a method can be defined to work on values of type T, independently of which T will be chosen by the user of the method
- Java's **Generics** support parametric polymorphism
  - Used to build data structures for objects of any class

Java Generics

- In version 1.5, Java introduced Generics
- Generics implement parametric polymorphism
  - Allow to define classes and methods containing and operating on values of a generic type T, where T will be chosen by the user of the class or method
- Syntax
  - Type parameters are enclosed in angle brackets: `<T>`
  - Multiple type parameters are possible: `<T,U,V>`

Generics and Primitives

- Generics require classes
  - A generic type parameter T represents a class
  - In Java, primitive types (int, char, byte, float, double) aren't classes
- Can't use primitive types with Generics

```
ListNode<int> n = new ListNode<int>();
```

```
GenericList.java:8: error: unexpected type
        ListNode<int> b = new ListNode<int>();
                         ^
  required: reference
  found:    int
```

Wrapper Classes

- Wrapper classes
  - To allow storing primitive values in collections, Java has wrapper classes like `Integer`, `Double`, `Character`, etc.

```
  ListNode<Integer> n = new ListNode<Integer>();
```

  - Contain a primitive value and offer various convenience methods and constructors

```
Integer aObject = new Integer(1);
int aPrimitive = Integer.intValue();
```

- Wrapped objects
  - Instances of wrapper classes are objects, not primitives

```
Integer a = new Integer(1);
Integer b = new Integer(1);
System.out.println(a == b);  // Prints false, should use
                             // equals() instead!
```

Autoboxing

Autoboxing and Autounboxing refers to the automatic conversion between primitive types and their corresponding wrapper classes.

The Java compiler automatically inserts code (calls to intValue() and valueOf()) to convert between primitive types and wrapper classes

- Danger
  - Autoboxing can hide the fact you're dealing with objects instead of integers and lead to hard-to-spot bugs!

```
ListStack<Integer> n = new ListStack<Integer>();
n.push(12345);
n.push(12345);
System.out.println(n.pop() == n.pop()); // Prints false
```